PROJECT

Software Engineering Department

SOFTWARE CONSTRUCTION & DEVELOPMENT

Submitted by

ABDUL MOIZ | 65315

Lab Instructor

MAAM LAILA

Session Fall – 2023

BALOCHISTAN UNIVERSITY OF INFORMATION TECHNOLOGY, ENGINEERING AND MANAGEMENT SCIENCES, QUETTA.

1. Introduction

1.1 Project Description

This project is a **Smart Inventory & Warehouse Management System**, developed using **Node.js, Express, MySQL, and React.js** as part of the Software Construction & Design (SCD) course.

The main goal of this project is to demonstrate **practical usage of Software Design Patterns** inside a real-world inventory system.
The system allows users to:

- Manage warehouse items
- Track stock levels
- Add new items
- Register suppliers
- View low-stock alerts
- Generate dashboard summaries

Design Patterns Used:

- **Factory Pattern** → For item creation
- **Composite Pattern** → For warehouse structure (Warehouse → Sections → Racks → Bins)
- **Decorator Pattern** → For adding extra metadata to items (Fragile, Heavy, Critical)
- **Observer Pattern** → For auto-trigger low-stock alerts
- **Facade Pattern** → For consolidated reporting (summary stats, alert counts)

This project fully implements Object-Oriented concepts and SCD design principles.

2. Design Pattern Usage

2.1 Factory Pattern

Problem It Solves:

System needs to create items with different configurations (id, name, sku, category) without repeating object construction code.

Implementation:

ItemFactory returns fully structured item objects simply by passing required fields.

Class-Level Structure:

- ItemFactory → Creates new items
- Returned Object → Contains id, sku, quantity, category, labels

UML (Text Format):

ItemFactory → ItemObject

2.2 Composite Pattern

Problem It Solves:

Warehouse structure has multiple nested levels:
Warehouse → Sections → Racks → Bins.
Managing them individually is difficult.

Implementation:

Composite classes: WarehouseGroup, WarehouseComponent, Section, Rack, Bin.

Class Structure:

```
WarehouseGroup
 ├─ Section
 |   ├─ Rack
 |   |   └─ Bin
```

2.3 Decorator Pattern

Problem It Solves:

Items need extra metadata like **Fragile**, **Critical**, **Heavy**, but we shouldn't create subclasses for every combination.

Implementation:

Decorators wrap items and add new metadata.

Classes:

- ItemDecorator (base)
- FragileDecorator
- CriticalDecorator
- HeavyDecorator

2.4 Observer Pattern

Problem It Solves:

When an item goes below minimum stock, system should automatically create alerts.

Implementation:

Observers watch inventory changes and trigger alerts when conditions match.

Class Structure:

- InventoryObserver
- AlertObserver
- Database table: alerts

2.5 Facade Pattern

Problem It Solves:

Dashboard needs combined statistics from multiple database tables.

Implementation:

ReportingFacade joins multiple MySQL queries and returns a single structured summary.

Facade Outputs:

- Total items

- Unique SKUs
- Low-stock items
- Supplier count
- Alerts by severity

3. Implementation Details

3.1 Technologies Used

Frontend:

- React.js + Functional Components
- Fetch API for backend communication
- CSS (Dark UI theme)

Backend:

- Node.js
- Express.js
- MySQL Database
- dotenv
- uuid
- CORS

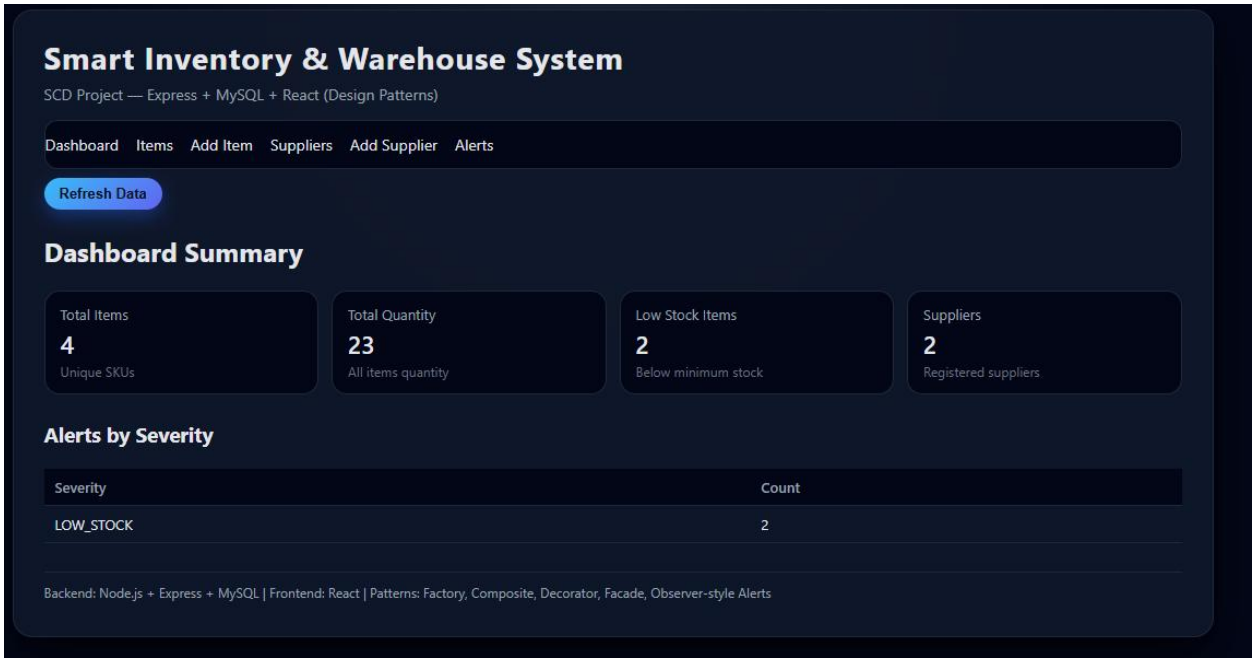
Database:

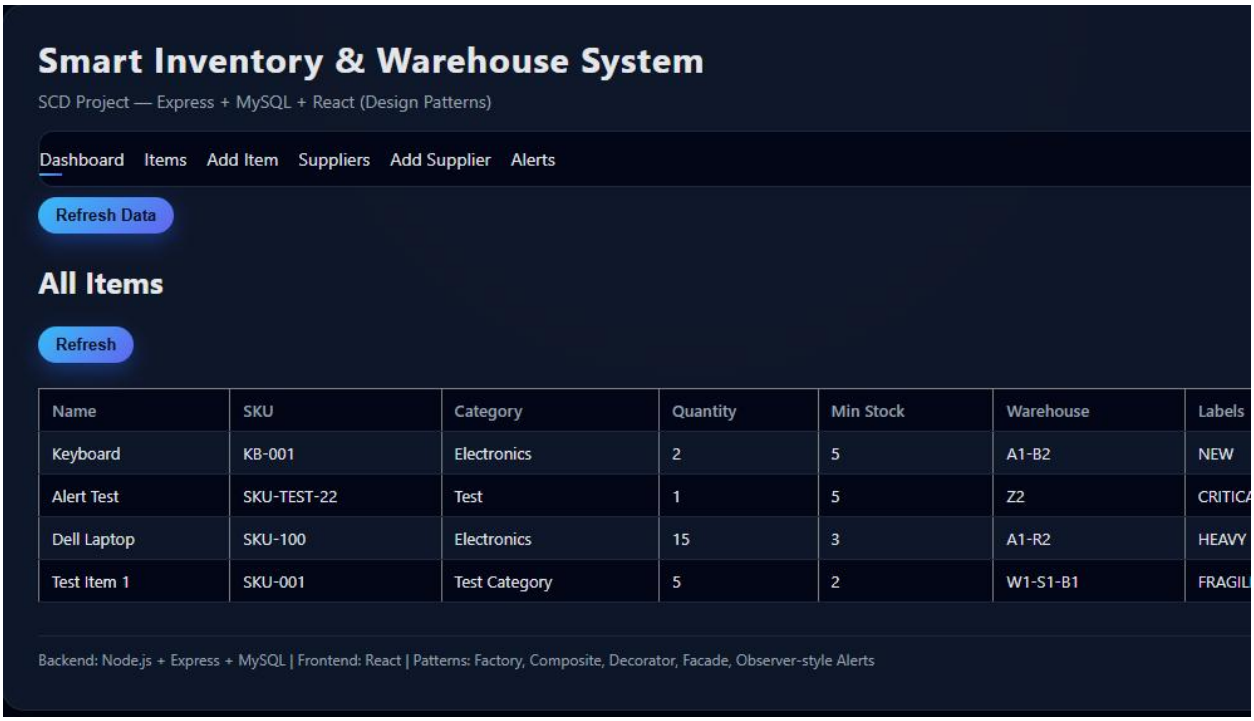
Tables created:

- items
- suppliers
- supplier_items
- alerts

4. Screenshots / Output

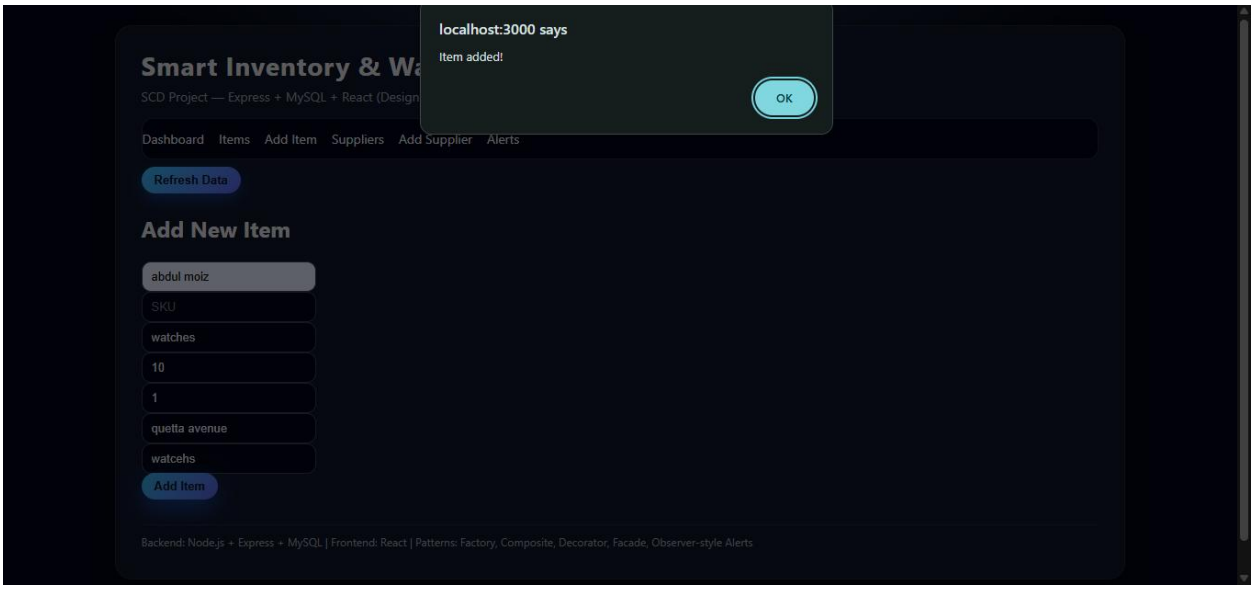
1. Dashboard view



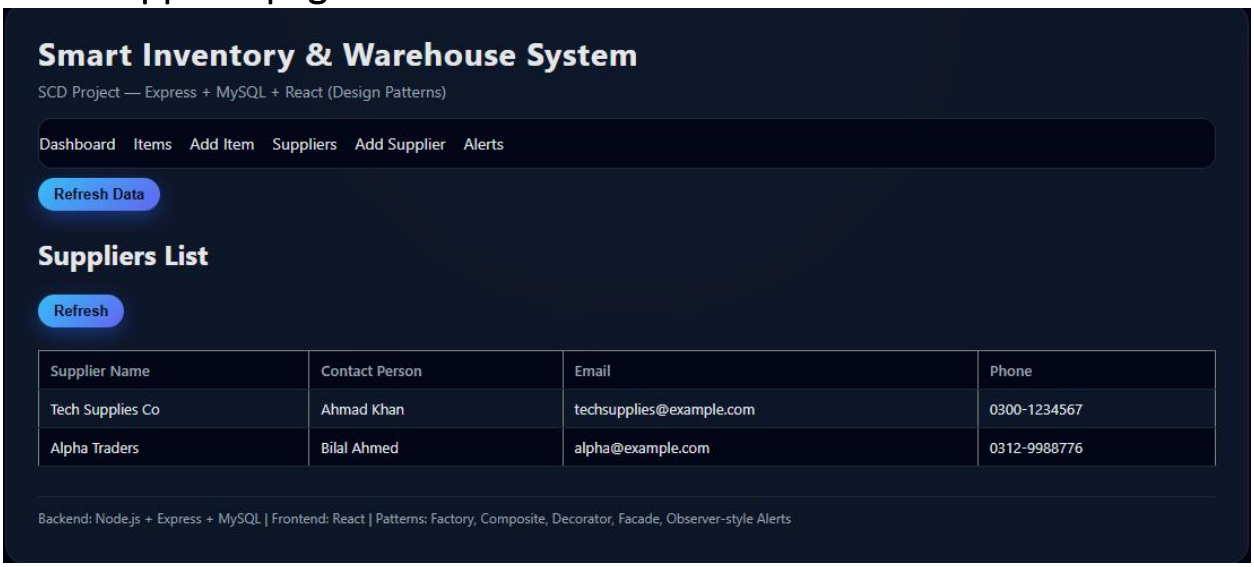


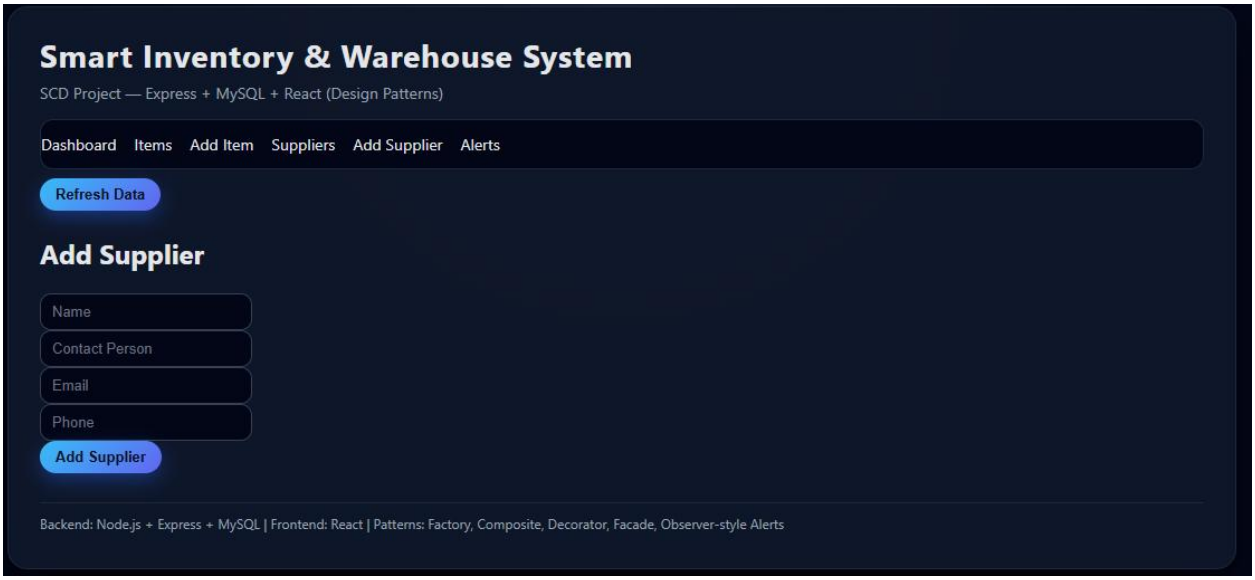
2. Items list

3. Add Item form



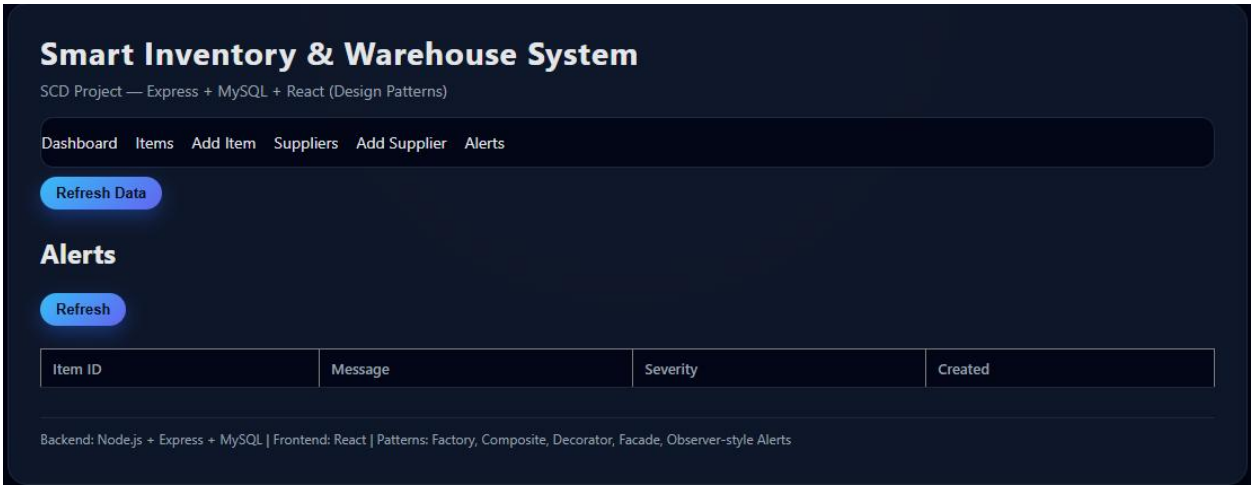
4. Suppliers page





5. Add supplier form

6. Alerts page



Problems Faced & Solutions

Problem 1: MySQL connection errors & environment variables

Solution: Used .env + proper MySQL credentials + error handling.

Problem 2: Auto-refresh alerts not triggering

Solution: Implemented Observer pattern and database triggers within itemService.

Problem 3: Hard to build warehouse structure

Solution: Composite pattern solved hierarchical structure cleanly.

Problem 4: Repeated logic for item creation

Solution: Implemented Factory pattern for clean object creation.

Problem 5: Dashboard required multiple SQL joins

Solution: Added Reporting Facade for combined results.

Conclusion

This project successfully demonstrates how software design patterns improve system structure, maintainability, and scalability.

By using
Factory, Composite, Decorator, Observer, and Facade Patterns,
the system becomes modular and easy to extend.

The backend (Node + Express + MySQL) and frontend (React.js) work together to provide a complete Smart Inventory Management solution.

This project fulfills all SCD course requirements with proper pattern implementations, clean UI, and professional system architecture.